Aufgabe 1: Arukone

Team-ID: 00889

Team: Felix Haag

Bearbeiter/-innen dieser Aufgabe:
Felix Haag

12. November 2023

Inhaltsverzeichnis

Lösungsidee

In der Aufgabe sind für die Erstellung eines Arukone-Rätsels mehrere Regeln gegeben: Jeder Linienzug besteht nur aus horizontalen und vertikalen Abschnitten (nicht schräg) (Regel 1), in jedem Feld mit einer Zahl beginnt oder endet genau ein Linienzug (Regel 2) und jedes Feld ohne Zahl wird von genau einem Linienzug durchlaufen oder ist leer (Regel 3). Um ein lösbares Arukone-Rätsel zu erstellen, bei dem die Zahlenpaare nicht direkt nebeneinander liegen, müssen allerdings noch weitere Regeln definiert werden: Zwei gleiche Zahlen sollten nicht in benachbarten Feldern liegen (Regel 4) und jedes Feld mit einer Zahl braucht mindestens ein leeres benachbartes Feld (Regel 5).

Zuerst wird ein Gitter der Größe n x n erstellt und eine Anzahl an Zahlenpaaren ausgewählt. Diese muss nach der Aufgabenstellung mindestens n/2 sein und ist maximal n.

Für die erste Zahl jedes Zahlenpaares kann ein zufälliges Feld im Gitter gewählt. Jedes Feld, auf dem eine Zahl platziert wird, muss nach Regel 4 und 5 mindestens zwei freie benachbarte Felder in eine Richtung haben. Beim Platzieren jeder Zahl muss dies deshalb überprüft werden. Zusätzlich muss noch überprüft werden, ob alle bereits platzierten Zahlen nach Platzieren der Zahl auf diesem Feld weiterhin mindestens zwei freie benachbarte Felder in eine Richtung hätten. Ist beides gegeben, kann die Zahl auf diesem Feld platziert werden, wenn nicht muss ein neues zufällig ausgewählt werden.

Beim Platzieren der zweiten Zahl jedes Zahlenpaares wird versucht ein Feld möglichst weit weg von der ersten auszuwählen, um zu vermeiden, dass die Zahlenpaare zu nah beieinander liegen. Es

werden alle noch freien Felder vom am weitesten entfernten bis zum nächsten durchgegangen. Für jedes dieser Felder wird unter Beachtung von Regel 1 bis 3 der schnellste Weg gesucht, bei dem nach Regel 4 und 5 weiterhin alle anderen einzelnen Zahlen mindestens zwei freie benachbarte Felder in eine Richtung haben. Wird ein solcher Weg gefunden, wird die Zahl auf diesem Feld platziert.

Umsetzung

Die Lösungsidee wird in Python implementiert. Zuerst wird für das Gitter eine zwei-dimensionale Liste der Größe n x n gefüllt mit Nullen erstellt. Jedes Feld des Gitters entspricht somit einem Wert der Liste an der Stelle (y, x), $0 < y, x < n$. Mit der random-Bibliothek wird eine zufällige Anzahl an Zahlenpaaren zwischen n/2 und n ausgewählt. Jede Zahl 1 bis Anzahl der Zahlenpaare wird in einer zweiten Liste gespeichert und die Liste dann zufällig durchgemischt.

Zum Platzieren der ersten Zahlen jedes Zahlenpaares auf dem Gitter wird nun die Liste der Zahlen mit einer for-Schleife durchlaufen. Für jede Zahl werden zuerst in einer weiteren Liste alle Positionen, an denen in der Gitter-Liste noch keine Zahl oder ein Linienzug (dargestellt durch -1) vorhanden ist, gespeichert. Hierfür wird die Gitter-Liste mit zwei for-Schleifen durchlaufen und jeder Wert überprüft. Ist der Wert 0, ist das Feld noch frei. Diese Liste wird ebenfalls zufällig durchgemischt.

Mit einer while-schleife werden alle Positionen der Liste mit freien Feldern durchlaufen. Bei jedem Durchlauf wird das vorderste, ein durch die Durchmischung zufälliges Element der Liste entnommen. Es wird überprüft, ob noch alle Zahlen mindestens zwei leere benachbarte Felder haben, wenn die Zahl auf das Feld dieser Position gesetzt werden würde. Wenn ja, wird die Zahl auf dem Feld platziert, wenn nicht wird das nächste überprüft.

Zum Platzieren der zweiten Zahl jedes Zahlenpaares wird die Liste der Zahlen wieder zufällig durchmischt und wieder alle Positionen, an denen in der Gitter-Liste noch keine Zahl oder ein Linienzug vorhanden ist, in einer Liste gespeichert. Diese Liste wird dann mit dem Satz des Pythagoras nach der größten Entfernung zur Partnerzahl sortiert.

Anschließend werden wieder mit einer while-schleife alle Positionen der Liste durchlaufen. Bei jedem Durchlauf wird wieder das vorderste Element der Liste entnommen. Zuerst wird wieder überprüft, ob noch alle Zahlen mindestens zwei leere benachbarte Felder haben, wenn die Zahl auf das Feld dieser Position gesetzt werden würde. Danach wird ein Weg zwischen dem aktuell untersuchten Feld und dem Feld der Partnerzahl gesucht. Dabei kommt der Lee-Algorithmus zum Einsatz, der den schnellsten Weg zwischen den beiden Feldern sucht (weiteres unten). Wird ein Weg gefunden, wird die Zahl auf dem aktuellen Feld platziert und alle Linienzüge mit -1 in der Gitter-Liste eingetragen. Wenn nicht, wird mit der nächsten Position fortgefahren.

Sind alle Zahlen im Gitter platziert, werden alle Linienzüge wieder aus dem Gitter entfernt, um ein leeres zu lösendes Arukone-Rätsel zu erhalten. Zum Schluss wird die Größe des Gitters, die Anzahl der Zahlenpaare und das Gitter selbst in der Konsole ausgegeben und in einer Textdatei gespeichert.

Finden des schnellsten Weges zwischen zwei Feldern

Zum finden des schnellsten Weges zwischen Zwei Felder wird eine Klasse „Node“ erstellt, die für jeden Wegpunkt bei der Suche des Weges die Koordinaten und eine Referenz zum vorherigen Wegpunkt speichert. In einem Enum sind alle möglichen Richtungen (hoch, rechts, runter, links) gespeichert.

Es wird eine Funktion definiert, die die Gitter-Liste, die Position des aktuell untersuchten Feldes, die Zahl, die platziert werden soll und die Anzahl der Zahlenpaare übergeben bekommt. Für den Algorithmus wird eine Queue verwendet, die alle als nächstes zu betrachtenden Wegpunkte beinhaltet. Zu dieser wird zuerst der Startknoten mit den Koordinaten des Startfeldes hinzugefügt und das Startfeld in der dafür vorgesehen Liste als besucht markiert.

Nun wird die Queue mit einer while-Schleife durchlaufen, solange bis sie leer ist. Bei jedem Durchlauf wird der vorderste Knoten der Queue entnommen. Danach werden alle Richtungen des Direction-Enums mit einer for-Schleife durchlaufen. Es werden die neuen x-, y- und z-Koordinaten berechnet, die nach Gehen in die jeweilige Richtung entstehen würden. Dafür werden die Werte der Richtung aus dem Enum zu den Koordinaten des entnommenen Knotens aus der Queue addiert. Zuerst wird überprüft, ob die neuen Koordinaten bereits dem gesuchten Ende entsprechen. Wenn ja, gibt die Funktion einen neuen Knoten mit diesen Koordinaten und Referenz zum vorherigen Knoten zurück. Ansonsten wird überprüft ob das Feld an den neuen Koordinaten im Gitter noch frei ist, also keine Zahl oder Linienzug enthält. Anschließend wird noch überprüft ob nach dem Platzieren des Linienzuges auf diesem Feld noch alle anderen einzelnen Zahlen mindestens zwei freie benachbarte Felder in eine Richtung hätten. Trifft dies zu, wird das Feld als besucht markiert und ein neuer Knoten mit den neuen Koordinaten und Referenz zum vorherigen Knoten erstellt und zur Queue hinzugefügt. Ist die Queue leer, wurde kein Weg gefunden und es wird „None“ zurückgegeben.

Beispiele

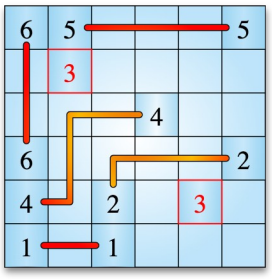
arukone0.txt

Größe 6 x 6, 6 Zahlenpaaren

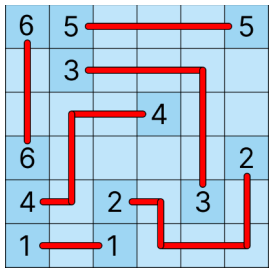
Ausgabe:

```
6
6
6 5 0 0 0 5
0 3 0 0 0 0
0 0 0 4 0 0
6 0 0 0 0 2
4 0 2 0 3 0
1 0 1 0 0 0
```

Arukone-Checker:



Lösung:



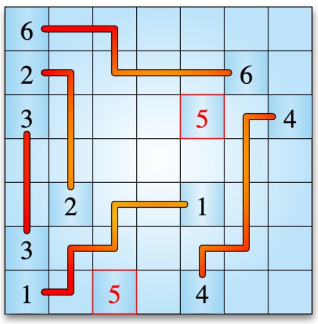
arukone1.txt

Größe 7 x 7, 6 Zahlenpaaren

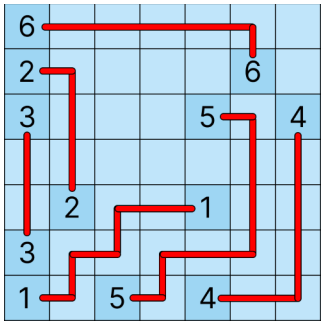
Ausgabe:

```
7
6
6 0 0 0 0 0 0
2 0 0 0 0 6 0
3 0 0 0 5 0 4
0 0 0 0 0 0 0
0 2 0 0 1 0 0
3 0 0 0 0 0 0
1 0 5 0 4 0 0
```

Arukone-Checker:



Lösung:

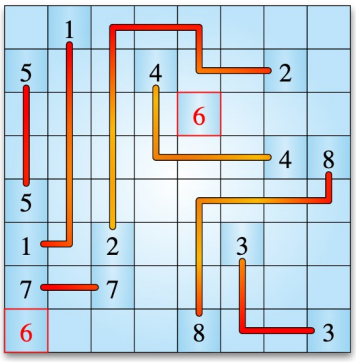


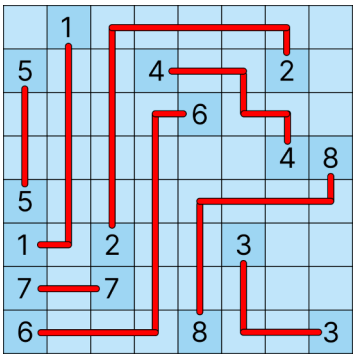
arukone2.txt

Größe 8 x 8, 8 Zahlenpaaren

Ausgabe: Arukone-Checker: Lösung:

```
8
8
0 1 0 0 0 0 0 0
5 0 0 4 0 0 2 0
0 0 0 0 6 0 0 0
0 0 0 0 0 0 4 8
5 0 0 0 0 0 0 0
1 0 2 0 0 3 0 0
7 0 7 0 0 0 0 0
6 0 0 0 8 0 0 3
```





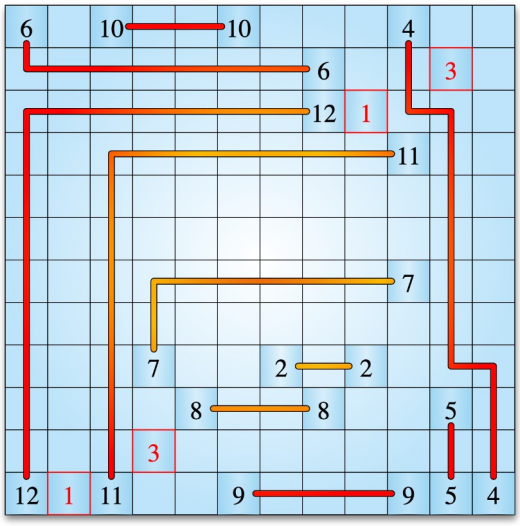
arukone3.txt

Größe 12 x 12, 12 Zahlenpaaren

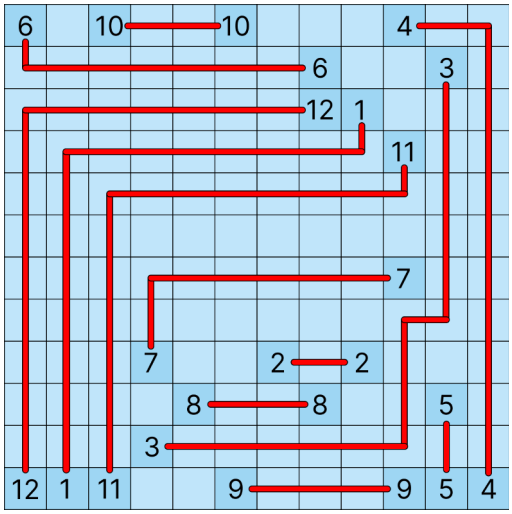
Ausgabe:

```
12
12
6 0 10 0 0 10 0 0 0 4 0 0
0 0 0 0 0 0 6 0 0 3 0
0 0 0 0 0 0 12 1 0 0 0
0 0 0 0 0 0 0 0 11 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 7 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 7 0 0 2 0 2 0 0 0
0 0 0 0 8 0 0 8 0 0 5 0
0 0 0 3 0 0 0 0 0 0 0 0
12 1 11 0 0 9 0 0 0 9 5 4
```

Arukone-Checker:



Lösung:



Quellcode

Arukone Generieren

```
def generate_arukone(n, number_count):
    grid = [[0 for _ in range(n)] for _ in range(n)] # Gitter der Größe n x n

    if number_count == 0:
        number_count = randint(n // 2, n) # Anzahl der Zahlen von n/2 bis n

    numbers = [] # Liste mit allen Zahlen
    for i in range(1, number_count+1): # Jede Zahl wird zwei mal hinzugefügt
        numbers.append(i)

    shuffle(numbers) # Zahlen werden zufällig durchgemischt

    for number in numbers:
        posible_positions = []

        for r, row in enumerate(grid):
            for c, col in enumerate(row):
                if col == 0:
                   # Alle Positionen an denen noch keine Zahl oder Linienzug vorhanden ist werden zu den
                     möglichen Positionen hinzugefügt
                    posible_positions.append((c, r))

        shuffle(posible_positions)

        while len(posible_positions) > 0:
            x, y = posible_positions.pop(0) # Zufällige mögliche Position wird der Liste entnommen

            new_grid = deepcopy(grid) # Kopie des Gitters
            new_grid[y][x] = number # Hinzufügen der Zahl zur Gitter-Kopie

            # Überprüfung ob jetzt noch alle Zahlen mindestens ein leeres benachbartes Feld haben
            if numbers_have_space(new_grid, number_count):
                grid[y][x] = number # Hinzufügen der Zahl zum Gitter
                break # Weiter mit nächster Zahl

            continue
```

```
    shuffle(numbers)

    for number in numbers:
        posible_positions = []

        for r, row in enumerate(grid):
            for c, col in enumerate(row):
                if col == 0:
                    # Alle Positionen an denen noch keine Zahl oder Linienzug vorhanden ist werden zu den
                      möglichen Positionen hinzugefügt
                    posible_positions.append((c, r))

        # Position der Zahl, falls diese bereits einmal im Gitter vorhanden ist
        number_pos = find_position(grid, number)

        posible_positions.sort(key= lambda p: math.hypot(p[1]-number_pos[1], p[0]-number_pos[0]),
                               reverse=True)

        for x, y in posible_positions:

            # Überprüfung ob Position direkt neben der bereits vorhandenen Zahl ist
            if (number_pos[1] == y and abs(number_pos[0]-x) <= 1) or (number_pos[0] == x and
                abs(number_pos[1]-y) <= 1):
                continue # Wenn nicht, weiter mit nächster möglicher Position

            # Finden des schnellsten Weges von der Zahl zu der bereits vorhandenen gleichen Zahl
            end_node = find_path(grid, number, x, y, number_count)
            if end_node == None: # Kein Weg gefunden
                continue # Weiter mit nächster möglicher Position
            else: # Weg gefunden
                new_grid = deepcopy(grid) # Kopie des Gitters
                new_grid[y][x] = number # Hinzufügen der Zahl zur Gitter-Kopie

                # Überprüfung ob jetzt noch alle Zahlen mindestens ein leeres benachbartes Feld haben
                if not(numbers_have_space(new_grid, number_count)):
                    continue # Wenn nicht, weiter mit nächster möglicher Position

                grid[y][x] = number # Hinzufügen der Zahl zum Gitter

                prev_node = end_node.previous_node
                while prev_node.previous_node != None: # Durchlaufen des Weges von Zahl zu Zahl
                    grid[prev_node.y][prev_node.x] = -1 # Eintragen der Linienzügen im Gitter
                    prev_node = prev_node.previous_node

    # Entfernen der Linienzüge aus dem Gitter
    for r in range(len(grid)):
        for c in range(len(grid[r])):
            if grid[r][c] == -1:
                grid[r][c] = 0

    print(n) # Ausgeben der Größe des Gitters
    print(number_count) # Ausgeben der Anzahl an Zahlen
    print_grid(grid) # Ausgeben des Gitters

    save_grid(n, number_count, grid) # Speichern des Rätsels in Textdatei
```

Finden des schnellsten Weges zwischen zwei Zahlen

```
# Wegpunkt bei der Suche eines Weges
class Node:
    def __init__(self, x, y, previous_node):
        self.x = x
        self.y = y
        self.previous_node = previous_node # vorheriger Wegpunkt des Weges

# Alle Richtungen in die man von einem feld aus gehen kann
class Direction(Enum):
    UP = (0, -1)
    RIGHT = (1, 0)
    DOWN = (0, 1)
    LEFT = (-1, 0)

# Lee-Algorithmus
def find_path(grid, number, x, y, number_count):
    # Zu findendes Ende des Weges ist die Position der bereits vorhandenen Zahl im Gitter
    end = find_position(grid, number)

    queue = [] # Queue in der alle zu gehenden Wegpunkte gespeichert werden

    # Startpositions wird zur Queue hinzugefügt, Node speichert Position, sowie vorherige Node des Weges
    queue.append(Node(x, y, None))

    # Kopie des Gitters wird erstellt, discovered beinhaltet, welche Felder beim Suchen des Weges schon
      besucht wurden
    discovered = deepcopy(grid)
    discovered[y][x] = number # Startposition wird als besucht markiert

    while(not(len(queue) == 0)):
        node = queue.pop(0) # Vorderste Node wird aus Queue entfernt

        # Jede Richtung (oben, rechts, unten, links) wird durchlaufen
        for dir in Direction:
            newX = node.x + dir.value[0] # neuer y-Wert in jeweilige Richtung
            newY = node.y + dir.value[1] # neuer x-Wert in jeweilige Richtung

            if newX == end[0] and newY == end[1]: # Ende wurde gefunden
                return Node(newX, newY, node) # Letzter Schritt des Weges wird zurückgegeben

            # Überprüfung ob neue Position innerhalb des Gitters liegt, an dieser Position noch keine
              andere Zahl oder Linienzug vorhanden ist und diese Position noch nicht besucht wurde
            if newY >= 0 and newY < len(grid) and newX >= 0 and newX < len(grid[newY]) and (grid[newY]
               [newX] == 0 or grid[newY][newX] == number) and (discovered[newY][newX] == 0 or
               discovered[newY][newX] == number):
                new_discovered = deepcopy(discovered)
                new_discovered[newY][newX] = -1

                # Überprüfung ob nach setzten des Linienzuges noch alle Zahlen Platz haben
                if numbers_have_space(new_discovered, number_count):
                    discovered[newY][newX] = -1 # Position wird als besucht markiert
                    queue.append(Node(newX, newY, node)) # Wegpunkt wird zur Queue hinzugefügt

    return None
```

Überprüfung ob alle Zahlen Platz haben

```
def number_has_space(grid, number):
    number_count = 0 # Anzahl, wie oft eine Zahl im Gitter vorkommt
    for r, row in enumerate(grid):
        for c, col in enumerate(row):
            if col == number:
                number_count += 1 # Zahl kommt im Gitter vor

    # Falls die Zahl noch nicht oder schon zwei mal im Gitter vorhanden ist, muss nichts weiter Überprüft
     werden
    if number_count == 0 or number_count == 2:
        return True
    # Falls Zahl erst einmal im Gitter vorhanden ist, muss sie mindestens zwei leere benachbarte Felder in
     eine Richtung haben, um Platz für die zweite Zahl zu haben
    elif number_count == 1:
        x, y = find_position(grid, number) # Position der Zahl im Gitter finden
        directions = { "up": [False, (x, y-1), (x, y-2)], "right": [False, (x+1, y), (x+2, y)], "down":
                     [False, (x, y+1), (x, y+2)], "left": [False, (x-1, y), (x-2, y)] }
        # zwei leere benachbarte Felder nach oben?
        directions["up"][0] = y-1 >= 0 and y-2 >= 0 and grid[y-1][x] == 0 and grid[y-2][x] == 0
         # zwei leere benachbarte Felder nach rechts?
        directions["right"][0] = x+1 < len(grid[y]) and x+2 < len(grid[y]) and grid[y][x+1] == 0 and
                                 grid[y][x+2] == 0
       # zwei leere benachbarte Felder nach unten?
        directions["down"][0] = y+1 < len(grid) and y+2 < len(grid) and grid[y+1][x] == 0 and
                                grid[y+2][x] == 0
       # zwei leere benachbarte Felder nach links?
        directions["left"][0] = x-1 >= 0 and x-2 >= 0 and grid[y][x-1] == 0 and grid[y][x-2] == 0

        # Falls Zahl mindestens zwei leere banchbarte Felder in eine Richtung hat
        if directions["up"][0] or directions["right"][0] or directions["down"][0] or directions["left"][0]:
            possible_directions = {k: v for k, v in directions.items() if v[0] == True}

            keys = list(possible_directions.keys())

            shuffle(keys)

            direction = possible_directions[keys[0]] # Zufällige freie Richtung ausgewählt

            # Felder in diese Richtung als belegt markieren
            grid[direction[1][1]][direction[1][0]] = -1
            grid[direction[2][1]][direction[2][0]] = -1

            return True

    return False

def numbers_have_space(grid, number_count):
    # Durchlaufen aller Zahlen im Gitter
    for i in range(1, number_count+1):
       # Überprüfung ob Zahl mindestens ein leeres benachbartes Feld im Gitter hat
        has_space = number_has_space(grid, i)
        if not(has_space):
            return False

    return True
```